

On Summarizing Graph Streams

Nan Tang Qing Chen* Prasenjit Mitra
Qatar Computing Research Institute
{ntang, qchen, pmitra}@qf.org.qa

ABSTRACT

Graph streams, which refer to the graph with edges being updated sequentially in a form of a stream, have wide applications such as cyber security, social networks and transportation networks. This paper studies the problem of *summarizing graph streams*. Specifically, given a graph stream G , directed or undirected, the objective is to summarize G as S_G with much smaller (*sublinear*) space, *linear* construction time and *constant* maintenance cost for each edge update, such that S_G allows many queries over G to be approximately conducted efficiently. Due to the sheer volume and highly dynamic nature of graph streams, summarizing them remains a notoriously hard, if not impossible, problem. The widely used practice of summarizing data streams is to treat each element independently by *e.g.*, hash- or sampling-based method, without keeping track of the connections between elements in a data stream, which gives these summaries limited power in supporting complicated queries over graph streams. This paper discusses a fundamentally different philosophy for summarizing graph streams. We present **gLava**, a probabilistic graph model that, instead of treating an edge (a stream element) as the operating unit, uses the finer grained node in an element. This will naturally form a new graph sketch where edges capture the connections inside elements, and nodes maintain relationships across elements. We discuss a wide range of supported graph queries and establish theoretical error bounds for basic queries.

1. INTRODUCTION

Massive graphs arise in many applications *e.g.*, network traffic data, social networks, citation graphs, and transportation networks. These graphs are highly dynamic: Network traffic data averages to about 10^9 packets per hour per router for large ISPs [18]; Twitter sees 100 million users login daily, with around 500 millions tweets per day [1].

Example 1: We consider a graph stream¹ as a sequence of elements $(x, y; t)$, which indicates that the edge (x, y) is encountered at time t . A graph stream $\langle (a, b; t_1), (a, c; t_2), \dots (b, a; t_{14}) \rangle$ is depicted in Fig. 1, where all timestamps are omitted for simplicity. Each edge is associated with a weight, which is 1 by default. \square

Queries over big graph streams have many important applications, *e.g.*, monitoring cyber security attack and traffic networks. Unfortunately, in a recent survey for businesses in analyzing big data streams [2], although 41% of respondents stated that the ability to analyze and act on streaming data in minutes was critical, 67% also admitted that they do not have the infrastructure to support that goal. The situation is more complicated and challenging when handling graph streams. It poses unique space and time constraints on storing, summarizing, maintaining and querying graph streams, due to its sheer volume, high velocity, and the complication of various queries.

In fact, for the applications of data streams, fast and approximated answers are often preferred than exact answers. Hence, sketch synopses have been widely studied for estimations over data streams. Although they allow false positives, the space savings normally prevail over this drawback, when the probability of an error is sufficiently low. There exist many sketches: AMS [5], Lossy Counting [23], CountMin [10], Bottom- k [9] and **gSketch** [31]. They have also been used in commercial tools, *e.g.*, CountMin is in Cisco OpenSOC² cyber security analytics framework.

Example 2: Consider the graph stream in Example 1 and Fig. 1. CountMin treats each element in the stream independently, and maps them to w hash buckets, by using the pairs of node labels as hash keys. Assume there are $w = 4$ hash buckets. The set of values to be hashed is $\{ab, ac, bc, \dots\}$, where the value ab is to concatenate two node labels for the element $(a, b; t_1)$. A hash function is used to put these values in 4 buckets, *i.e.*, $h(\cdot) \rightarrow [1, 4]$, as shown in Fig. 2. The first bucket aggregates the weights of values starting with a and e , and similar for other buckets. Note that, for simplicity, we use only one hash function for illustration. In fact, CountMin uses d pairwise independent hash functions to alleviate the problem of hash key collisions.

CountMin, and its variant **gSketch**, can be used for estimating certain types of graph queries. (a) *Edge query*. It can estimate the weight of particular edges: 5 for ab , and 1

*Qing is a QCRI intern, from Fudan University, China.

¹Without loss of generality, we use a directed graph for illustration. Our method can also work on undirected graphs.

²<http://opensoc.github.io/>

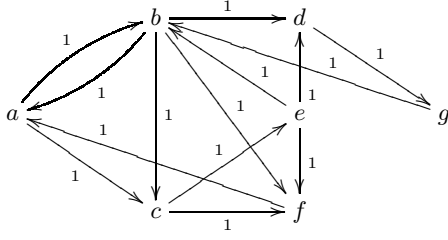


Figure 1: A sample graph stream

$h(\cdot)$	5	5	3	1
	ab, ac, ed, eb, ef	bc, bd, ba, bf, fa	ce, cf, gb	dg

Figure 2: A hash-based sketch

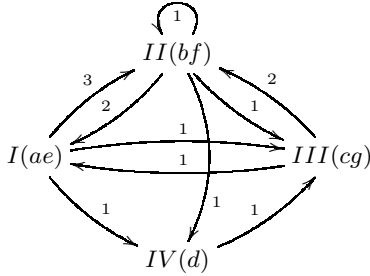


Figure 3: An example of our proposed sketch

for dg . (b) *Aggregate subgraph query*. It can answer queries like: *What is the aggregated weight for a graph with two edges (a, c) and (c, e) ?* It will give $5 + 3 = 8$ as the estimated weight for the graph. \square

Example 2 shows the applicability of CountMin on some query estimation over graph streams. However, their (and other sketch synopses') main weakness is that they are element (or edge) independent. That is, they fall short of maintaining the connections between streaming elements, thus fail in estimating many important queries, such as node monitoring, path queries (e.g., reachability) and more complicated graph analytics (see Section 3.4 for a detailed discussion).

Challenges. Designing a graph sketch to support a wide range of applications requires to satisfy the following constraints. (1) *Space constraint*: *sublinear* upper bound is needed. (2) *Time constraint*: *linear* construction time is required. Noticeably, this is stronger than the constraint with only a constant passes over the stream. (3) *Maintenance constraint*: to maintain it for one element insertion/deletion should be in *constant* time. (4) *Element connectivity*: the *connectivity* between elements should be maintained.

To this end, we present **gLava**, a novel generalized graph sketch to meet the above required constraints. Instead of treating elements (edges) in a graph stream independently as those prior art, the key idea of **gLava** is to compress a graph stream based on a finer grained item, the node in a stream element.

Example 3: Again, consider the graph stream in Fig. 1. Our proposed sketch is shown in Fig. 3. For each edge $(x, y; t)$, **gLava** uses a hash function to map each node label

to 4 node buckets i.e., $h(\cdot) \rightarrow [1, 4]$. Node I is the summary of two node labels a and e , assuming $h(a) = 1$ and $h(e) = 1$. The other compressed nodes are computed similarly. The edge weight denotes the aggregated weights from stream elements, e.g., the number 3 from node I to II means that there are three elements as $(x, y; t)$ where the label of x (resp. y) is a or e (resp. b or f). \square

Remark. (1) It is readily to see that estimation for edge frequencies, as what **CountMin** supports, can be easily achieved by **gLava**. Better still, the idea of using multiple hash functions to alleviate hash collisions can be easily applied to **gLava**. (2) **gLava** is represented as a graph, which captures not only the connections inside elements, but also the links across elements. These make it an ideal sketch to support a much wider range of applications, compared with those prior art (see Section 3.4 for a discussion).

Contributions. This paper presents a novel graph sketch for supporting a wide range of graph stream applications.

(1) We introduce **gLava**, a novel graph sketch (Section 3.3). As shown in Fig. 3, the proposed sketch naturally preserves the graphical connections of the original graph stream, which makes it a better fit than traditional data stream sketches to support analytics over graph streams. We further categorize its supported graph queries (Section 3.4).

(2) We describe algorithms to process various graph analytics on top of **gLava** (Section 4). The general purpose is, instead of proposing new algorithms, to show that **gLava** can easily be used to support many graph queries, and *off-the-shelf* graph algorithms can be seamlessly integrated.

(3) We perform theoretical analysis to establish error bounds for basic queries, specifically, the edge frequency estimation and point query estimation (Section 5).

(4) We describe implementation details (Section 6). This is important to ensure that the new sketch can be constructed and maintained under the hard time/space constraints to support streaming applications. Moreover, we propose to use non-square matrices, using the same space, to improve the accuracy of estimation.

In this history of graph problems over streams, unfortunately, most results showed that a large amount of space is required for *complicated* graph problems [4, 24]. The present study fills a gap in the literature by analyzing various graph problems in a small amount of space. We thus contend that **gLava** will shed new light on graph stream management.

Organization. Section 2 discusses related work. Section 3 defines the new sketch and queries to be solved. Section 4 describes algorithms using **gLava**. Section 5 makes theoretical analysis. Section 6 describes implementation details. Finally, Section 7 concludes the paper with a summary of our findings.

2. RELATED WORK

We categorize related work as follows.

Sketch synopses. Given a data stream, the aim of sketch synopses is to apply (linear) projections of the data into lower dimensional spaces that preserve the salient features of the data. A considerable amount of literature has been published on general data streams such as AMS [5], lossy counting [23], **CountMin** [10] and bottom- k [9]. The work

gSketch [31] improves CountMin for graph streams, by assuming that data samples or query samples are given. Bloom filters have been widely used in a variety of network problems (see [7] for a survey). There are also sketches that maintain counters only for nodes, *e.g.*, using a heap for maintaining the nodes the largest degrees for heavy hitters [11].

As remarked earlier, gLava is more general, since it maintains connections inside and across elements, *without* assuming any sample data or query is given. None of existing sketches maintains both node and edge information.

Graph summaries. Summarizing graphs has been widely studied. The most prolific area is in web graph compression. The papers [3, 27] encode Web pages with similar adjacency lists using reference encoding, so as to reduce the number of bits needed to encode a link. The work [25] groups Web pages based on a combination of their URL patterns and k -means clustering. The paper [13] compresses graphs based on specific types of queries. There are also many clustering based methods from data mining community (see *e.g.*, [19]), with the basic idea to group *similar* nodes together.

These data structures are designed for less dynamic graphs, which are not suitable for graph stream scenarios.

Graph pattern matching over streams. There have been several work on matching graph patterns over graph streams, based on either the semantics of subgraph isomorphism [8, 15, 30] or graph simulation [26]. The work [30] assumes that queries are given, and builds node-neighbor tree to filter false candidate results. The paper [15] leverages a distributed graph processing framework, Giraph, to approximately evaluate graph queries. The work [8] uses the subgraph distributional statistics collected from the graph streams to optimize a graph query evaluation. The paper [26] uses filtering methods to find data that potentially matches for a specific type of queries, namely *degree-preserving dual simulation with timing constraints*.

Firstly, all the above algorithms are designed for a certain type of graph queries. Secondly, most of them assume the presence of queries, so they can build indices to accelerate. In contrast, gLava aims at summarizing graph streams in a generalized way, so as to support various types of queries, *without* any assumption of queries.

Graph stream algorithms. There has also been work on algorithms over graph streams (see [24] for a survey). This includes the problems of connectivity [14], trees [28], spanners [12], sparsification [20], counting subgraphs *e.g.*, triangles [6, 29]. However, they mainly focus on theoretical study for best approximation bound, mostly on $O(n \text{ polylog } n)$ space, with one to multiple passes over the data stream.

In fact, gLava is a friend, instead a competitor, of them. As will be seen later (Section 4), gLava can treat existing algorithms as *black-boxes* to help solve existing problems.

Distributed graph systems. Many distributed graph computing systems have been proposed to conduct data processing and data analytics in massive graphs, such as Pregel [22], Giraph³, GraphLab [21], Power-Graph [16] and GraphX [17].

They have been proved to be efficient on static graphs, but are not ready for doing analytics over big graph streams with real-time response. In fact, they are complementary to and can be used for gLava in distributed settings.

3. GLAVA AND SUPPORTED QUERIES

We first define graph streams (Section 3.1) and state the studied problem (Section 3.2). We then introduce our proposed graph sketch model (Section 3.3). Finally, we discuss the queries supported by our new sketch (Section 3.4).

3.1 Graph Streams

A *graph stream* is a sequence of elements $e = (x, y; t)$ where x, y are node identifiers (labels) and edge (x, y) is encountered at time-stamp t . Such a stream,

$$G = \langle e_1, e_2, \dots, e_m \rangle$$

naturally defines a graph $G = (V, E)$ where V is a set of nodes and $E = \{e_1, \dots, e_m\}$. We write $\omega(e_i)$ the weight for the edge e_i , and $\omega(x, y)$ the aggregated edge weight from node x to node y . We call m the *size* of the graph stream, denoted by $|G|$.

Intuitively, the node label, being treated as an identifier, uniquely identifies a node, which could be *e.g.*, IP addresses in network traffic data or user IDs in social networks. Note that, in the graph terminology, a graph stream is a *multi-graph*, where each edge can occur many times, *e.g.*, one IP address can send multiple packets to another IP address. We are interested in properties of the underlying graph. The causes a main challenge with graph streams where one normally does not have enough space to record the edges that have been seen so far. Summarizing such graph streams in one pass is important to many applications. Moreover, we do not explicitly differentiate whether (x, y) is an ordered pair or not. In other words, our approach applies naturally to both directed and undirected graphs.

For instance, in network traffic data, a stream element arrives at the form: $(192.168.29.1, 192.168.29.133, 62, 105.12)^4$, where node labels 192.168.29.1 and 192.168.29.133 are IP addresses, 62 is the number of bytes sent *from* 192.168.29.1 to 192.168.29.133 in this captured packet (*i.e.*, the weight of the directed edge), and 105.12 is the time in seconds that this edge arrived when the server started to capture data. Please see Fig. 1 for a sample graph stream.

3.2 Problem statement

The problem of *summarizing graph streams* is, given a graph stream G , to design another data structure S_G from G , such that:

1. $|S_G| \ll |G|$: the size of S_G is far less than G , preferably in sublinear space.
2. The time to construct S_G from G is in linear time.
3. The update cost of S_G for each edge insertion/deletion is in constant time.

Intuitively, a graph stream summary has to be built and maintained in real time, so as to deal with big volume and high velocity graph stream scenarios. In fact, the CountMin [23] and its variant gSketch [31] satisfy the above conditions (see Example 2 for more details). Unfortunately, as discussed earlier, CountMin and gSketch can support only limited types of graph queries.

³<http://giraph.apache.org>

⁴We omit port numbers and protocols for simplicity.

3.3 The gLava Model

The graph sketch. A *graph sketch* is a graph $S_G(\mathcal{V}, \mathcal{E})$, where \mathcal{V} denotes the set of vertices and \mathcal{E} its edges. For vertex $v \in \mathcal{V}$, we simply treat its label as its node identifier (the same as the graph stream model). Each edge e is associated with a weight, denoted as $\omega(e)$.

In generating the above graph sketch S_G from a graph G , we first set the number of nodes in the sketch, *i.e.*, let $|\mathcal{V}| = w$. For an edge $(x, y; t)$ in G , we use a hash function h to map the label of each node to a value in $[1, w]$, and the aggregated edge weight is calculated correspondingly.

Please refer to Fig. 3 as an example, where we set $w = 4$.

Discussion about edge weight. The edge weight for an edge e in the graph sketch is computed by an *aggregation* function of all edge weights that are mapped to e . Such an aggregation function could be $\min(\cdot)$, $\max(\cdot)$, $\text{count}(\cdot)$, $\text{average}(\cdot)$, $\text{sum}(\cdot)$ or other functions. In this paper, we use $\text{sum}(\cdot)$ by default to explain our method. The other aggregation functions can be similarly applied. In practice, which aggregation function to use is determined by applications. For a more general setting that requires to maintain multiple aggregated functions in a graph sketch, we may extend our model to have multiple edge weights *e.g.*, $\omega_1(e), \dots, \omega_n(e)$, with each $\omega_i(e)$ corresponds to an aggregation function.

Remark. One may observe that the graph sketch model is basically the same as the graph stream model, with the main difference that the time-stamps are not maintained. This makes it very useful and applicable in many scenarios when querying a graph stream for a given time window. In other words, for a graph analytics method M that needs to run on a streaming graph G , denoted as $M(G)$, one can run it on its sketch S_G , *i.e.*, $M(S_G)$, to get an estimated result, without modifying the method M .

Example 4: Consider the graph stream in Fig 1 and its sketch in Fig. 3. Assume that query Q_1 is to estimate the aggregated weight of edges from b to c . In Fig. 3, one can map b to node II , c to node III , and get the estimated weight 1 from edge (II, III) , which is precise. Now consider Q_2 , which is to compute the aggregated weight from g to b . One can locate the edge (III, II) , and the estimated result is 2, which is not accurate since the real weight of (g, b) in Fig 1 is 1. \square

The above result is expected, since given the compression, no hash function can ensure that the estimation on the sketch can be done precisely. Along the same line of CountMin [10], we use multiple independent hash functions to reduce the probability of hash collisions.

The gLava model. A gLava sketch is a set of graph sketches $\{S_1(\mathcal{V}_1, \mathcal{E}_1), \dots, S_d(\mathcal{V}_d, \mathcal{E}_d)\}$. Here, we use d hash functions h_1, \dots, h_d , where h_i ($i \in [1, d]$) is used to generate S_i . Also, h_1, \dots, h_d are chosen uniformly at random from a pairwise-independent family (see Section 6.2 for more details).

Example 5: Figure 4 shows another two sketches for Fig. 1. Again, consider the query Q_2 in Example 4. Using S_1 in Fig. 4 (a), one can locate edge (III, II) for (g, b) , which gives 1. Similarly, S_2 in Fig. 4 (b) will also give 1 from edge (iii, i) , where g (resp. b) maps to iii (resp. i). The minimum of the above two outputs is 1, which is correct. \square

Example 5 shows that using multiple hash functions can indeed improve the accuracy of estimation.

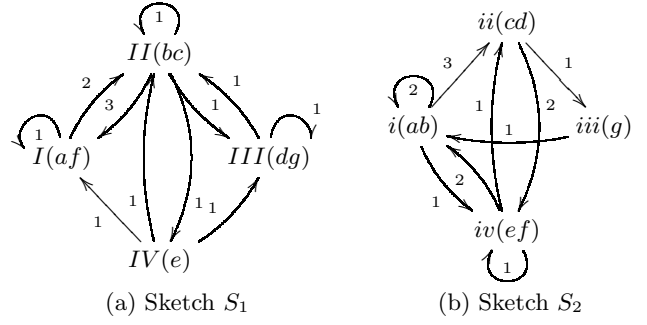


Figure 4: A gLava sketch with 2 hash functions

3.4 Supported Graph Queries

As remarked in Section 3.3, for any graph analytics method M to run over G , *i.e.*, $M(G)$, it is possible to run M on each sketch directly and individually, and then merge the result as: $\hat{M}(G) = \Gamma(M(S_1), \dots, M(S_d))$, where $\hat{M}(G)$ denotes the estimated result over G , and $\Gamma(\cdot)$ an aggregation function (*e.g.*, \min , \max , conjunction) to merge results returned from d sketches.

Whilst the exercise in this section only substantiates several types of graph queries to be addressed in this work, it is evident that the power of gLava is far beyond what are listed below.

Edge frequency. Given two node labels a and b , we write $f_e(a, b)$ to denote the exact weight from a a -node to a b -node. We write $\tilde{f}_e(a, b)$ the estimated weight from a sketch.

One application of such queries, taking social networks for example, is to estimate the communication frequency between two specific friends.

Point queries. For a directed graph, given a node label a , we study a boolean query $f_v(a, \leftarrow) > \theta$ (resp. $f_v(a, \rightarrow) < \theta$), which is to monitor whether the aggregated edge weight *to* (resp. *from*) a node with label a is above (resp. below) a given threshold θ in the graph stream G . For an undirected graph, we write $f_v(a, \perp) > \theta$ and $f_v(a, \perp) < \theta$ correspondingly. Similarly, we use $\tilde{f}_v(a, \leftarrow)$, $\tilde{f}_v(a, \rightarrow)$, $\tilde{f}_v(a, \perp)$ for estimated numbers using sketches.

One important application of such queries is DoS (Denial-of-service) attacks in cyber security, which typically flood a target source (*i.e.*, a computer) with massive external communication requests.

Path queries. Given two node labels a and b , a (boolean) reachability query $r(a, b)$ is to tell whether they are connected. Also, we write $\tilde{r}(a, b)$ as the estimated reachability.

One important monitoring task, for the success of multicast deployment in the Internet, is to verify the availability of service in the network, which is usually referred to as *reachability monitoring*. Another reachability application that needs to consider edge weights is IP routing, which is to determine the path of data flows in order to travel across multiple networks.

Aggregate subgraph query. The *aggregate subgraph query* is considered in gSketch [31]. It is to compute the aggregate weight of the constituent edges of a subgraph $Q = \{(x_1, y_1), \dots, (x_k, y_k)\}$, denoted by $f(Q) = \Omega(f_e(x_1, y_1), \dots, f_e(x_k, y_k))$. Here, the function $\Omega(\cdot)$ is to merge weights from all $f_e(x_i, y_i)$ for $i \in [1, k]$. We write

Symbols	Notations
G, S_G	a graph stream, and a graph sketch
$\omega(e)$	weight of the edge e
$f_e(a, b)$	edge weight
$f_v(a, \leftarrow)$	node in-flow weight (directed graphs)
$f_v(a, \rightarrow)$	node out-flow weight (directed graphs)
$f_v(a, \perp)$	node flow weight (undirected graphs)
$r(a, b)$	whether b is reachable from a
$f(Q)$	weight of subgraph Q

Table 1: Notations

$\tilde{f}(Q)$ for estimation over sketches.

Note that, gSketch simply merges all estimated weights in G , *e.g.*, the sum of all weights, even if one edge is missing (*i.e.*, $\tilde{f}(x_i, y_i) = 0$ for some edge e_i). However, we use a different query semantics in this work: if $\tilde{f}(x_i, y_i) = 0$, the estimated aggregate weight should be 0, since the query graph Q does not have an exact match. We use this revised semantics since it is a more practical setting.

Example 6: Consider a subgraph with two edges as $Q : \{(a, b), (a, c)\}$. The query $Q_3 : \tilde{f}(Q)$ is to estimate the aggregate weight of Q . The precise answer is 2, which is easy to check from Fig. 1. \square

Extensions. We consider an extension of the above aggregate subgraph query, which allows a wildcard $*$ in the node labels that are being queried. More specifically, for the subgraph query $Q = \{(x_1, y_1), \dots, (x_k, y_k)\}$, each x_i or y_i is either a constant value, or a wildcard $*$ (*i.e.*, match any label). A further extension is to *bound* the wildcards to be matched to the same node, by using a subscript to a wildcard as $*_j$. That is, two wildcards with the same subscripts enforce them to be mapped to the same node.

Example 7: A subgraph query $Q_4 : \tilde{f}(\{(a, b), (b, c), (c, a)\})$ is to estimate the triangle, *i.e.*, a 3-clique with three vertices labeled as a, b , and c , respectively. Another subgraph query $Q_5 : \tilde{f}(\{(*, b), (b, c), (c, *)\})$ is to estimate paths that start at node with an edge to b , and end at any node with an edge from c , if the edge (b, c) exists in the graph. If one wants to count the common neighbors of (b, c) , the following query $Q_6 : \tilde{f}(\{(*_1, b), (b, c), (c, *_1)\})$ can enforce such a condition, which again is a case of counting triangles. \square

The extension is apparently more general, with the purpose to cover more useful queries in practice. Unfortunately, gSketch cannot support such extensions.

Summary of notations. The notations of this paper are summarized in Table 1, which are for both directed and undirected graphs, unless being specified otherwise.

4. QUERY PROCESSING

We consider the graph stream as G , and d graph sketches $\{S_1, \dots, S_d\}$ of G , where S_i is constructed using a hash function $h_i(\cdot)$. In this following, we discuss how to process different types of queries. Again, by default, we assume that we use $\text{sum}(\cdot)$ as the default aggregation function. Also, to facilitate the discussion, we assume that an adjacent matrix is used for storing gLava.

4.1 Edge Query

To evaluate $\tilde{f}_e(a, b)$, the edge weight of (a, b) , is straight-

forward. It is to find the estimated edge weight from each sketch $\omega_i(h_i(a), h_i(b))$ and then use a corresponding function $\Gamma(\cdot)$ to merge them, as the following:

$$\tilde{f}_e(a, b) = \Gamma(\omega_1(h_1(a), h_1(b)), \dots, \omega_d(h_d(a), h_d(b)))$$

In this case, the function $\Gamma(\cdot)$ is to take the minimum.

Complexity. It is easy to see that estimating the aggregate weight of an edge query is in $O(d)$ time and in $O(d)$ space, where d is a constant.

4.2 Point Queries

Here, we only discuss the case $\tilde{f}_v(a, \leftarrow) > \theta$. That is, given a new element $(x, y; t)$, estimate in real-time whether the aggregate weight to node a is above the threshold θ . The other cases, *i.e.*, $\tilde{f}_v(a, \leftarrow) < \theta$, $\tilde{f}_v(a, \rightarrow) > \theta$, $\tilde{f}_v(a, \rightarrow) < \theta$, $\tilde{f}_v(a, \perp) > \theta$ and $\tilde{f}_v(a, \perp) < \theta$, can be similarly processed.

We use the following strategy to monitor $\tilde{f}_v(a, \leftarrow) > \theta$, given an incoming edge $e : (x, y; t)$. Note that if $y \neq a$, we simply update all d sketches in constant time, since this is not about an edge to a . Next, we only describe the case of edge $e : (x, a; t)$.

Step 1. [Estimate current in-flow.] We write $\tilde{f}_v^i(a, \leftarrow)$ the estimated in-flow from the i -th sketch. $\tilde{f}_v^i(a, \leftarrow)$ can be computed by first locating the column in the adjacent matrix corresponding to label a (*i.e.*, $h_i(a)$), and then sum up the values in that column, *i.e.*, $\tilde{f}_v^i(a, \leftarrow) = \sum_{j=1}^w \mathcal{M}_i[j][h_i(a)]$. Here, w is the width of the adjacent matrix \mathcal{M}_i . Then,

$$\tilde{f}_v(a, \leftarrow) = \Gamma(\tilde{f}_v^1(a, \leftarrow), \dots, \tilde{f}_v^w(a, \leftarrow))$$

In this case, the function $\Gamma(\cdot)$ is to take the minimum.

Step 2. [Monitor e .] We only send an alarm, if $\tilde{f}_v(a, \leftarrow) + \omega(e) > \theta$.

Step 3. [Update all sketches.] Update all d sketches by aggregating the new edge weight $\omega(e)$.

Complexity. It is easy to see that step 1 takes $O(d + w)$ time and $O(d)$ space, where both d and w are constants.

4.3 Path Queries

We consider the reachability query $\tilde{r}(a, b)$, which is to estimate whether b is reachable from a . For such queries, we treat any off-the-shelf algorithm $\text{reach}(x, y)$ as a *black-box* and show our strategy.

Step 1. [Map.] We invoke $\text{reach}_i(h_i[a], h_i[b])$ on the i -th sketch (for $i \in [1, d]$), to decide whether the mapped node $h_i[b]$ is reachable from the mapped node $h_i[a]$.

Step 2. [Reduce.] We merge individual results as follows:

$$\tilde{r}(a, b) = \text{reach}_1(h_1[a], h_1[b]) \wedge \dots \wedge \text{reach}_d(h_d[a], h_d[b])$$

That is, the estimated result is *true* only if the mapped nodes are reachable from *all* d sketches.

The complexity of the above strategy is determined by the algorithm $\text{reach}(\cdot)$.

4.4 Aggregate Subgraph Query

We next consider the aggregate subgraph query $\tilde{f}(Q)$, which is to compute the aggregate weight of the constituent edges of a sub-graph Q . The process is similar to the above path queries, by using any existing algorithm $\text{subgraph}(Q)$.

Step 1. [Map.] We first invoke `subgraph`(Q) at each sketch to find subgraph matches, and calculate the aggregate weight, denoted by $\text{weight}_i(Q)$ for the i -th sketch.

Step 2. [Reduce.] We merge individual results as follows:

$$\tilde{f}(Q) = \min(\text{weight}_1(Q), \dots, \text{weight}_d(Q))$$

Note that, running a graph algorithm on a sketch is only applicable to `gLava`. It is not applicable to `gSketch` since `gSketch` by nature is an array of frequency counting, without maintaining the graphical structure as `gLava` does. Also, in the case $\text{weight}_i(Q)$ from some sketch that says that a subgraph match does not exist, we can terminate the whole process, which provides chances for optimization.

Optimization. Recall that a subgraph Q is defined as constituent edges as $\{(x_1, y_1), \dots, (x_k, y_k)\}$ (Section 3.4). An alternative way of estimating the aggregate subgraph query is to first compute the minimum value of each value and sum them up. We denote this approach by $\tilde{f}'(Q)$ as follows:

$$\tilde{f}'(Q) = \sum_{i=1}^k \tilde{f}_e(x_i, y_i)$$

It is readily to see that $\tilde{f}'(Q) \leq \tilde{f}(Q)$. Recall that there are two extensions of the subgraph queries (Section 3.4). For the first extension that a wildcard $*$ is used, the above optimization can be used. For instance, the edge frequency of $\tilde{f}_e(x, *)$ for a directed graph is indeed $\tilde{f}_v(x, \rightarrow)$. For the second extension that multiple wildcards $*_i$ are used to bound to the same node, this optimization cannot be applied.

5. ERROR BOUNDS

We study the error bounds for two basic types of queries: edge queries and point queries. Although `gLava` is more general, we show that theoretically, it has the same error bounds as `CountMin`.

5.1 Edge Frequency

Our proof for the error bound of edge frequency queries is an adaption of the proof used in `CountMin` in Sec. 4.1 of [10].

Theorem 1: *The estimation $\tilde{f}_e(j, k)$ of the cumulative edge weight of the edge (j, k) has the following guarantees, $f_e(j, k) \leq \tilde{f}_e(j, k)$ with probability at least $1 - \delta$, s.t., $f_e(j, k) \leq \tilde{f}_e(j, k) + \epsilon * n$, where $f_e(j, k)$ is the exact answer to the cumulative edge weight of the edge (j, k) , n denotes the number of nodes, and the error in answering a query is within a factor of ϵ with probability δ^5 .* \square

PROOF. For any query edge $f_e(j, k)$, for each hash function h_i , by construction, for any stream edge $e : (j, k; t)$, where $\omega(e)$ is the edge weight that is added to $\text{count}(h_i(j), h_i(k))$. Therefore, by construction, the answer to $f_e(j, k)$ is less than or equal to $\min_i(\text{count}(h_i(j), h_i(k)))$ for $i \in [1, d]$ where d is the number of hash functions.

Consider two edges (j, k) and (l, m) . We define an indicator variable $I_{i,j,k,l,m}$, which is 1 if there is a collision between two distinct edges, i.e., $(j \neq l) \wedge (k \neq m) \wedge (h_i(j) = h_i(l) \wedge h_i(k) = h_i(m))$, and 0 otherwise. By pairwise independence of the hash functions,

$$E(I_{i,j,k,l,m}) = \Pr[(h_i(j) = h_i(l) \wedge h_i(k) = h_i(m))]$$

⁵The parameters ϵ and δ are usually set by the user.

$$\leq (1/\text{range}(h_i))^2 = \epsilon'^2 / e^2$$

where e is the base used to compute natural logarithm, and $\text{range}(h_i)$ is the number of hash buckets of function h_i . Define the variable $X_{i,j,k}$ (random over choices of h_i) to count the number of collisions with the edge (j, k) , which is formalized as $X_{i,j,k} = \sum_{l=1 \dots n, m=1 \dots n} I_{i,j,k,l,m} a_{l,m}$, where n is the number of nodes in the graph. Since all a_i 's are non-negative in this case, $X_{i,j,k}$ is non-negative. We write $\text{count}[i, h_i(j), h_i(k)]$ as the count in the hash bucket relative to hash function h_i . By construction, $\text{count}[i, h_i(j), h_i(k)] = f_e(j, k) + X_{i,j,k}$. So clearly, $\min_i(\text{count}[i, h_i(j), h_i(k)]) \geq f_e(j, k)$.

$$\begin{aligned} E(X_{i,j,k}) &= E(\sum_{l=1 \dots n, m=1 \dots n} I_{i,j,k,l,m} f_e(l, m)) \\ &\leq (\sum_{l=1 \dots n, m=1 \dots n} E(I_{i,j,k,l,m}) \cdot f_e(l, m)) \leq (\epsilon'/e)^2 * n \end{aligned}$$

by pairwise independence of h_i , and linearity of expectation.

Let, $\epsilon'^2 = \epsilon$. By the Markov inequality,

$$\begin{aligned} \Pr[\tilde{f}_e(j, k) > f_e(j, k) + \epsilon * n] &= \Pr[\forall_i. \text{count}[i, h_i(j), h_i(k)] > f_e(j, k) + \epsilon * n] \\ &= \Pr[\forall_i. f_e(j, k) + X_{i,j,k} > f_e(j, k) + \epsilon * n] \\ &= \Pr[\forall_i. X_{i,j,k} > \epsilon * E(X_{i,j,k})] < e^{-d} \leq \delta \end{aligned}$$

\square

Our algorithm generates the same number of collisions and the same error bounds under the same probabilistic guarantees as `CountMin` for edge frequency queries.

5.2 Point Queries

We first discuss the query for node out-degree, i.e., $f_v(a, \rightarrow)$. Consider the stream of edges $e : (a, *; t)$, i.e., edges from node a to any other node indicated by a wildcard $*$. Drop the destination (i.e., the wildcard) of each edge. The stream now becomes a stream of tuples $(a, \omega(t))$ where $\omega(t)$ is the edge weight from node a at time t . When a query is posed to find the out-degree of node a , `CountMin` [10] returns the minimum of the weights in different hash buckets as the estimation of the flow out of node a .

If the number of unique neighbors (i.e., connected using one hop ignoring the weights) is sought, we adapt the procedure above by replacing $\omega(t)$ with 1 in the stream above for all nodes. Clearly, by construction, the answer obtained is an over-estimation of the true out-degree because of two reasons: (a) collisions in the hash-buckets, and (b) self-collision, i.e., we do not know if an edge has been seen previously and thus count the outgoing edge again even if we have seen it.

The case for in-degree point queries is similar. To compute the in-degree of a node, we simply adapt the stream to create a one-dimensional stream as in the case above. Drop the source for each edge to create a stream. The variations in the case of out-degree for the total in-flow and the number of neighbors with in-links to a node can be estimated as in the case of out-degree outlined above.

The error estimates for point queries (see Sec. 4.1 of [10]) hold for these cases.

Lemma 5.2: *The estimated out-degree (in-degree) is within a factor of ϵ with probability δ if we use $d = \lceil \ln(1/\delta) \rceil$ rows of pair-wise independent hash functions and $w = \lceil e/\epsilon \rceil$.* \square

from \ to	$I(af)$	$II(bc)$	$III(dg)$	$IV(e)$
$I(af)$	1	2	0	0
$II(bc)$	3	1	1	1
$III(dg)$	0	1	1	0
$IV(e)$	1	1	1	0

Figure 5: The adjacent matrix of S_1

6. IMPLEMENTATION DETAILS

In this section, we first discuss the data structures used to implement **gLava** (Section 6.1). We then introduce the definition of pairwise independent hash functions (Section 6.2). We also discuss its potential extension in a distributed environment (Section 6.3).

6.1 Adjacent Matrix and Its Extension

Using hash-based methods, although it is evident that it only requires 1-pass of the graph stream to construct/update a **gLava**. However, the linear time complexity of constructing and updating a **gLava** depends on the data structures used. For example, the adjacent list may not be a fit, since searching a specific edge is not in constant time.

6.1.1 Adjacent Matrix

An adjacency matrix is a means of representing which nodes of a graph are adjacent to which other vertices.

Example 8: Consider the sketch S_1 in Fig. 4 (a) for example. Its adjacent matrix is shown in Fig. 5. \square

Example 8 showcases a directed graph. In the case of an undirected graph, it will be a symmetric matrix.

Construction. Consider a graph stream $G = \langle e_1, e_2, \dots, e_m \rangle$ where $e_i = (x_i, y_i; t_i)$. Given a number of nodes w and a hash function $h(\cdot) \rightarrow [1, w]$. We use the following strategy.

Step 1. [Initialization.] Construct a $w \times w$ matrix \mathcal{M} , with all values initialized to be 0.

Step 2. [Insertion of e_i .] Compute $h(x_i)$ and $h(y_i)$. Increase the value of $\mathcal{M}[h(x_i)][h(y_i)]$ by $\omega(e_i)$, the weight of e_i .

In the above strategy, step 1 will take constant time to allocate a matrix. Step 2 will take constant time for each e_i . Hence, the time complexity is $O(m)$ where m is the number of edges in G . The space used is $O(w^2)$.

Deletions. [Deletion of e_i .] Insertions have been discussed in the above step 2. For the deletion of an e_i that is not of interest (*e.g.*, out of a certain time window), it is simply to decrease the value of $\mathcal{M}[h(x_i)][h(y_i)]$ by $\omega(e_i)$ in $O(1)$ time.

Alternatively, one may consider to use an adjacent hash-list is to maintain, for each vertex, a list of its adjacent nodes using a hash table. Given an edge $e_i(x_i, y_i; t_i)$, two hash operations are needed: The first is to locate x_i , and the second is to find y_i from x_i 's hash-list. Afterwards, it updates the corresponding edge weight. Adjacent list is known to be suitable when graph is sparse. However, in terms of compressed graph in our case, as will be shown later in experiments, most sketches are relatively dense, which makes the adjacent matrix the default data structure to manage our graph sketches.

6.1.2 Using Non-Square Matrices

from \ to	$i(abcd)$	$ii(efg)$
$I(a)$	2	0
$II(b)$	3	1
$III(c)$	0	2
$IV(d)$	0	1
$IV(e)$	2	1
$IV(f)$	1	0
$IV(g)$	1	0

Figure 6: A non-square matrix

When using a classical square matrix for storing a sketch, we have an $n \times n$ matrix. Consider all edges from node a such as $(a, *)$. Using any hash function will inevitably hash all of these edges to the same row. For example, in Fig. 5, all edges $(a, *)$ will be hashed to the first row of the matrix.

When there is only one hash function to use, due to the lack of *a priori* knowledge of data distribution, it is hard to decide the right shape of a matrix. However, the application of d sketches provides us an opportunity to heuristically reduce the chance of combined hash collisions.

The basic idea is, instead of using an $n \times n$ matrix with one hash function, we use an $m \times p$ matrix with two hash functions: $h_1(\cdot) \rightarrow [1, m]$ on the *from* nodes and $h_2(\cdot) \rightarrow [1, p]$ on the *to* nodes.

Example 9: Consider the graph stream in Fig.1. Assume that we use two hash functions: $h_1(\cdot) \rightarrow [1, 7]$ and $h_2(\cdot) \rightarrow [1, 2]$. The non-square matrix is shown in Fig. 6. \square

In practice, when we can generate multiple sketches, we heuristics use matrices $n \times n$, $2n \times n/2$, $n/2 \times 2n$, $4n \times n/4$, $n/4 \times 4n$, etc. That is, we use matrices with the same sizes but different shapes.

6.2 Pairwise Independent Hash Functions

Here, we only borrow the definition of pairwise independent hash functions⁶, while relying on existing tools to implement them.

A family of functions $\mathcal{H} = \{h|h(\cdot) \rightarrow [1, w]\}$ is called a *family of pairwise independent hash functions* if for two different hash keys x_i, x_j , and $k, l \in [1, w]$,

$$\Pr_{h \leftarrow \mathcal{H}}[h(x_i) = k \wedge h(x_j) = l] = 1/w^2$$

Intuitively, when using multiple hash functions to construct sketches, the hash functions used should be pairwise independent in order to reduce the probability of hash collisions. Please refer to [10] for more details.

6.3 Discussion: Distributed Environment

In this paper, we mainly discuss how to implement **gLava** in a centralized environment. However, it is easy to observe that **gLava** can be easily applied to a distributed environment, since the construction and maintenance of each sketch is independent of each other. Assuming that we have d sketches in one computing node, when m nodes are available, we can use $d \times m$ pairwise independent hash functions, which may significantly reduce the probability of errors.

7. CONCLUSION

⁶<http://people.csail.mit.edu/ronitt/COURSE/S12/handouts/lec5.pdf>

We have proposed a new graph sketch for summarizing graph streams. We have demonstrated its wide applicability to many emerging applications.

8. REFERENCES

- [1] Tweet statistics. <http://expandedramblings.com/index.php/march-2013-by-the-numbers-a-few-amazing-twitter-stats/10/>.
- [2] Vitria. <http://www.vitria.com/solutions/streaming-big-data-analytics/benefits/>.
- [3] M. Adler and M. Mitzenmacher. Towards compressing web graphs. In *Data Compression Conference*, 2001.
- [4] C. C. Aggarwal, editor. *Data Classification: Algorithms and Applications*. CRC Press, 2014.
- [5] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. *J. Comput. Syst. Sci.*, 58(1):137–147, 1999.
- [6] V. Braverman, R. Ostrovsky, and D. Vilenchik. How hard is counting triangles in the streaming model? In *ICALP*, 2013.
- [7] A. Z. Broder and M. Mitzenmacher. Survey: Network applications of bloom filters: A survey. *Internet Mathematics*, 1(4):485–509, 2003.
- [8] S. Choudhury, L. B. Holder, G. C. Jr., K. Agarwal, and J. Feo. A selectivity based approach to continuous pattern detection in streaming graphs. In *EDBT*, 2015.
- [9] E. Cohen and H. Kaplan. Tighter estimation using bottom k sketches. *PVLDB*, 1(1), 2008.
- [10] G. Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *J. Algorithms*, 55(1):58–75, 2005.
- [11] G. Cormode and S. Muthukrishnan. Space efficient mining of multigraph streams. In *PODS*, 2005.
- [12] M. Elkin. Streaming and fully dynamic centralized algorithms for constructing and maintaining sparse spanners. *ACM Transactions on Algorithms*, 7(2):20, 2011.
- [13] W. Fan, J. Li, X. Wang, and Y. Wu. Query preserving graph compression. In *SIGMOD*, 2012.
- [14] J. Feigenbaum, S. Kannan, A. McGregor, S. Suri, and J. Zhang. On graph problems in a semi-streaming model. *Theor. Comput. Sci.*, 348(2-3):207–216, 2005.
- [15] J. Gao, C. Zhou, J. Zhou, and J. X. Yu. Continuous pattern detection over billion-edge graph using distributed framework. In *ICDE*, 2014.
- [16] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, 2012.
- [17] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. Graphx: Graph processing in a distributed dataflow framework. In *OSDI*, 2014.
- [18] S. Guha and A. McGregor. Graph synopses, sketches, and streams: A survey. *PVLDB*, 5(12), 2012.
- [19] A. K. Jain and R. C. Dubes. *Algorithms for Clustering Data*. Prentice-Hall, 1988.
- [20] J. A. Kelner and A. Levin. Spectral sparsification in the semi-streaming setting. *Theory Comput. Syst.*, 53(2):243–262, 2013.
- [21] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed graphlab: A framework for machine learning in the cloud. *PVLDB*, 5(8), 2012.
- [22] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, pages 135–146, 2010.
- [23] G. S. Manku and R. Motwani. Approximate frequency counts over data streams. In *VLDB*, 2002.
- [24] A. McGregor. Graph stream algorithms: a survey. *SIGMOD Record*, 43(1):9–20, 2014.
- [25] S. Raghavan and H. Garcia-Molina. Representing web graphs. In *ICDE*, 2003.
- [26] C. Song, T. Ge, C. X. Chen, and J. Wang. Event pattern matching over graph streams. *PVLDB*, 8(4), 2014.
- [27] T. Suel and J. Yuan. Compressing the graph structure of the web. In *Data Compression Conference*, 2001.
- [28] R. E. Tarjan. Data structures and network algorithms. In *SIAM*. 1983.
- [29] C. E. Tsourakakis, U. Kang, G. L. Miller, and C. Faloutsos. DOULION: counting triangles in massive graphs with a coin. In *SIGKDD*, 2009.
- [30] C. Wang and L. Chen. Continuous subgraph pattern search over graph streams. In *ICDE*, 2009.
- [31] P. Zhao, C. C. Aggarwal, and M. Wang. gSketch: On query estimation in graph streams. *PVLDB*, 5(3), 2011.